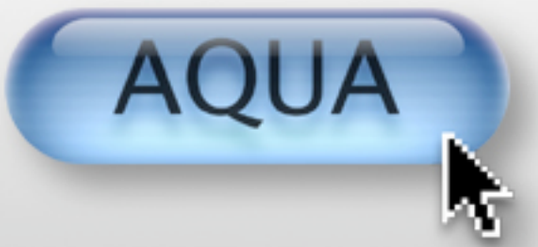
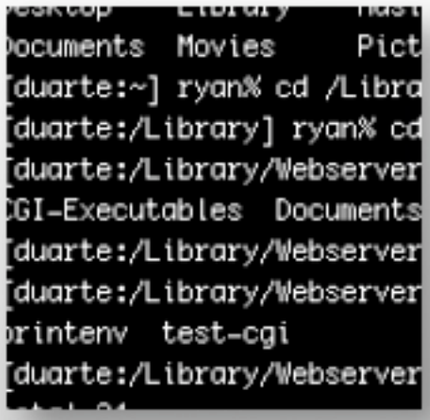
AppleScript: Making Your Carbon and Cocoa Application Scriptable

Session 414





AppleScript: Making Your Carbon and Cocoa Application Scriptable

John Comiskey
AppleScript

You Will Learn How to…

- Design a dictionary for a new or existing application
- Implement scriptable objects to effect that design
  - We will focus on Cocoa Scripting
  - With a little bit about Carbon at the end
- Test scriptable objects in a repeatable fashion

Designing a Dictionary

Designing an object model for scripting

Designing a Dictionary

Use an object model

- You use the power of OOP in your applications
- Let scripters have the same advantages
  - Reusable objects
    - Value added through leverage
  - Interoperability
    - Application suites become “SuperApplications”
  - Flexibility
    - “Why would anybody want to do that?”
  - New functionality
    - “AppleScript is the feature that you forgot”

Designing a Dictionary

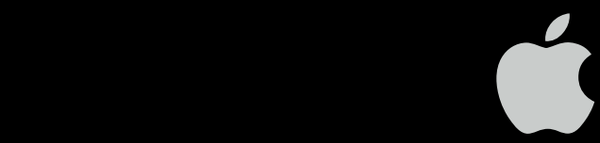
Use an object model

- AppleScript is an OOL
  - Inheritance, containment, data hiding, etc.
  - But only with an object model
- AppleScript is English-like
  - `verb object [clause] [clause] …`
  - But only with an object model

Designing a Dictionary

Use an object model

- Tell blocks are powerful and convenient
  - Scripts are less repetitive and easier to read
  - But only with an object model
- Applications can interoperate
  - Create synergies between applications
  - Script entire workflows with common objects
  - Wrap workflows in AppleScript Studio applications
  - But only with an object model



This Is NOT an Object Model

remove action from: Remove a folder action from a folder

```
remove action from reference  -- the object for the command
    [using action name Unicode text]  -- ...or the name of the action to remove
    [using action number integer]  -- the index number of the action to remove...
Result:  folder action  -- the reply for the command
```

do folder action: Send a folder action code to a folder action script

```
do folder action reference  -- the object for the command
    [with window size bounding rectangle]  -- the new window size for the folder action message to process
    [with item list anything]  -- a list of items for the folder action message to process
    folder action code items added/items removed/window closed/window moved/window opened  -- the folder action message to process
```

attached scripts: List the actions attached to a folder

```
attached scripts reference  -- the object for the command
Result:  list  -- the reply for the command
```

edit action of: Edit an action of a folder

```
edit action of reference  -- the object for the command
    [using action name Unicode text]  -- ...or the name of the action to edit
    [using action number integer]  -- the index number of the action to edit...
Result:  file  -- the reply for the command
```

attach action to: Attach an action to a folder

```
attach action to reference  -- the object for the command
    using Unicode text  -- a file containing the script to attach
Result:  folder action  -- the reply for the command
```

This Is an Object Model

Class folder action: An action attached to a folder in the file system
Plural form:
    folder actions
Elements:
    script by name, by numeric index, before/after another element, as a range of elements, satisfying a test
Properties:
    <Inheritance> item [r/o] -- *All of the properties of the superclass.*
    path anything [r/o] -- *the path to the folder to which the folder action applies*
    enabled boolean -- *Is the folder action enabled?*
    name Unicode text -- *the name of the folder action, which is also the name of the folder*
    volume Unicode text [r/o] -- *the volume on which the folder action resides*

Class script: A script invoked by a folder action
Plural form:
    scripts
Properties:
    <Inheritance> item [r/o] -- *All of the properties of the superclass.*
    name Unicode text [r/o] -- *the name of the script*
    POSIX path Unicode text [r/o] -- *the POSIX file system path of the disk*
    enabled boolean -- *Is the script enabled?*
    path Unicode text [r/o] -- *the file system path of the disk*

Designing a Dictionary

Use an object model

- Plan for a full object model
  - Map as much of the object hierarchy as possible
  - Decide now what future objects might be called
  - Define as much functionality as possible
- Implement as much as you can in the first release
  - Include enough to form a functional set
  - Leave the harder things as read-only
  - Provide at least one powerful read-write property

Designing a Dictionary

Use an object model

- Standard Suite verbs cover many needs
  - Implement as many as possible
    - `get` and `set` for properties
    - `exists`, `count`, `make`, and `delete` for elements
    - Six others for objects, and `quit`
  - Achieve a multiplicative effect
    - 3 objects x 10 properties x 2 verbs
    - 3 objects x 10 verbs
    - 90 separate operations, and `quit`

Designing a Dictionary

Don’t create a lot of verbs

- Procedural verbs are one dimensional
  - One operation performed on one data item
  - Every need must be anticipated
- Procedural verbs are ugly
  - Scripts are clumsy and hard to read
    - `remove action from “Some:Folder:Path” using action number 5`
  - There is a lot for the scripter to learn
- Your application’s scriptability is a dead end
  - Interoperability limited to native data types

Designing a Dictionary

Do create the verbs you really need

- Add value with verbs
  - Simple actions that your application performs
    - The names of certain buttons and menu items are verbs
  - Simple actions that the objects suggest
- Choose reusable verbs
  - Copy from other applications and suites
  - Keep terms and their four-byte-codes together
- Choose reusable parameters
  - Look in AERegistry.h and ASRegistry.h

Designing a Dictionary

Do create the verbs you really need

- Choose carefully
  - Use a property instead of a verb whenever possible

❌
```
enable script “Tim”
```

✅
```
set enabled of script “Tim” to true
```

  - Apply verbs to objects, rather than data
    - Exceptions usually involve an indirect object
    - `send “Hi, Bob!” to account “Bob Hartley”`
  - Avoid exposing raw API
    - Leads to one dimensional, procedural verbs

Designing a Dictionary

How do I choose the terms?

- Interview a user
  - Ask the user to describe what he's doing
  - Write down the steps he describes
    - That's an AppleScript!
  - Define the required objects and properties
  - Define new verbs only as necessary

Designing a Dictionary

How do I choose the terms?

- Consult the documentation
  - Use the same terms that the documentation uses
  - Look at an example or tutorial
    - That's an AppleScript!
  - Define the required objects and properties
  - Define new verbs only as necessary

Designing a Dictionary

How do I choose the terms?

- Consider your programmatic object model
  - Humanize the programmatic terms
  - Don’t expose everything
  - Don’t use InterCaps, use white space instead
  - Don’t use “is” in boolean properties
  - Use lower case, except for acronyms and proper nouns
    - `IP address`, `PostScript`, `POSIX path`
  - Define new verbs only as necessary

Designing a Dictionary

How do I choose the terms?

- Consider your GUI object model
  - Map screen items to AppleScript objects and properties
    - ❌ Don’t script the screen item itself
    - ✅ Do script the underlying property or element
  - Don’t just inventory everything, conceptualize
  - Omit articles and personal pronouns
  - Define new verbs only as necessary
    - The names of certain buttons and menu items are verbs

Designing a Dictionary

How do I choose the terms?

- Map screen items to objects and properties

numeric field	`integer`, `real`
check box	`boolean`
radio buttons	`enumeration`
small text field	`Unicode text`
large text field	`text` object

Designing a Dictionary

How do I choose the terms?

- Map screen items to objects and properties

some pop-up menus	`enumeration`
other pop-up menus	element relationship
some scrolling lists	`list`
other scrolling lists	element relationship
table views	element relationships

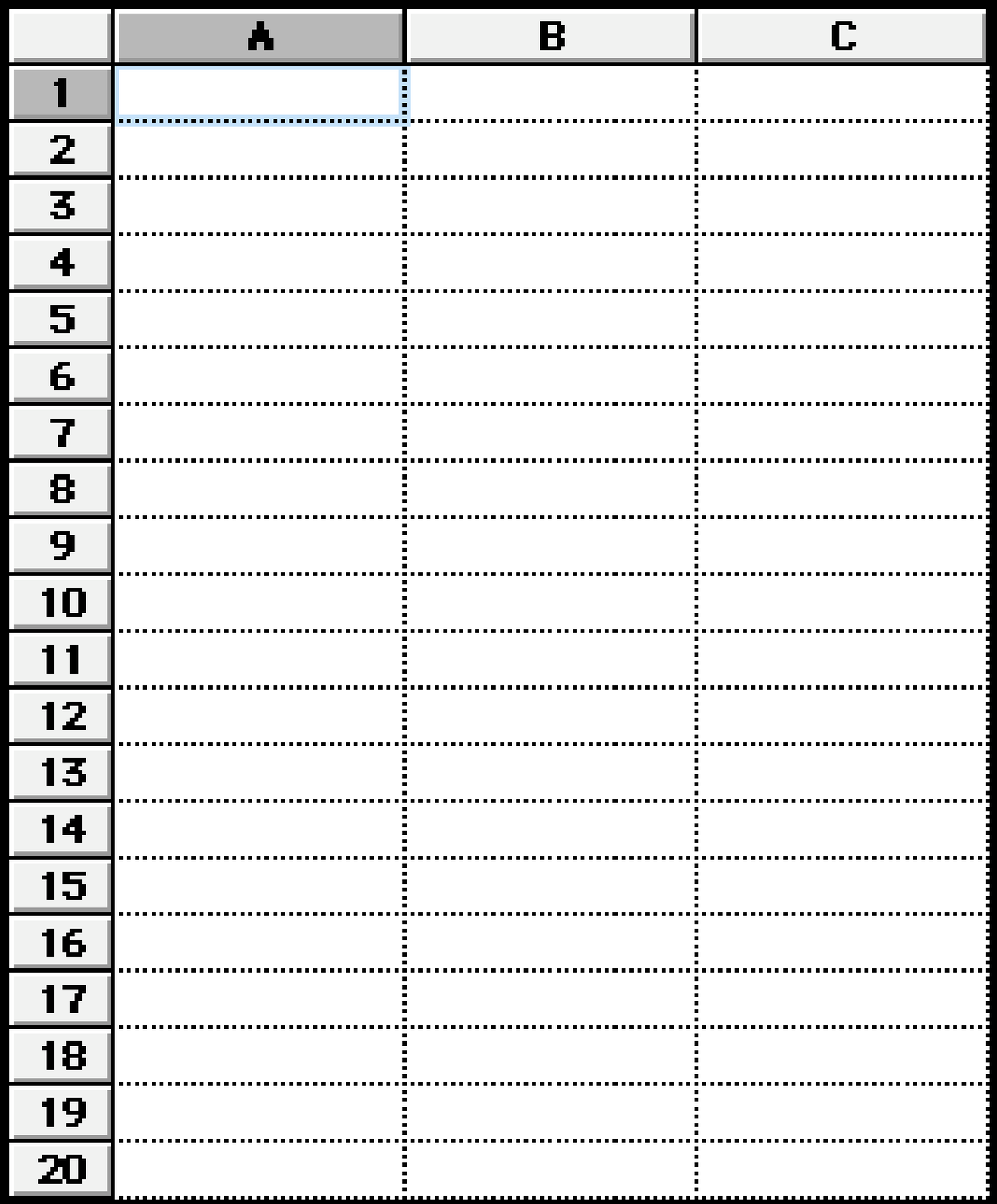


Table View

Physical mapping

- `table`
- `row`
  - By index
- `column`
  - By name
- `cell`
  - By name within row
  - By index within column
- `cells` contain objects

Table View

Abstract mapping

Last Name	First Name	City	State	Zip
Shula	Don	Grand River	OH	44045
Halas	George	Chicago	IL	60601
Landry	Tom	Mission	TX	78572
Lambeau	Curly	Green Bay	WI	54301
Noll	Chuck	Cleveland	OH	44101
Knox	Chuck	Sewickley	PA	15143
Reeves	Dan	Rome	GA	30149
Brown	Paul	Norwalk	OH	44857
Grant	Bud	Superior	WI	54880
Schottenheimer	Marty	Canonsburg	PA	15317
Levy	Marv	Chicago	IL	60601
Owen	Steve	Cleo Springs	OK	73729
Parcells	Bill	Englewood	NJ	07631
Gibbs	Joe	Mocksville	NC	27028
Stram	Hank	Gary	IN	46401
Ewbank	Weeb	Richmond	IN	47374
Ditka	Mike	Carnegie	PA	15106
Mora	Jim	Glendale	CA	91201
Seifert	George	San Francisco	CA	94101
Gillman	Sid	Minneapolis	MN	55401
Allen	George	Detroit	MI	48201
Coryell	Don	Seattle	WA	98101
Madden	John	Austin	MN	55912
Holmgren	Mike	San Francisco	CA	94101
Parker	Buddy	Slaton	TX	79364

- `database`
- `record`
  - By index, by unique id
- `field`
  - By name
  - Properties
    - `name`
    - `value`

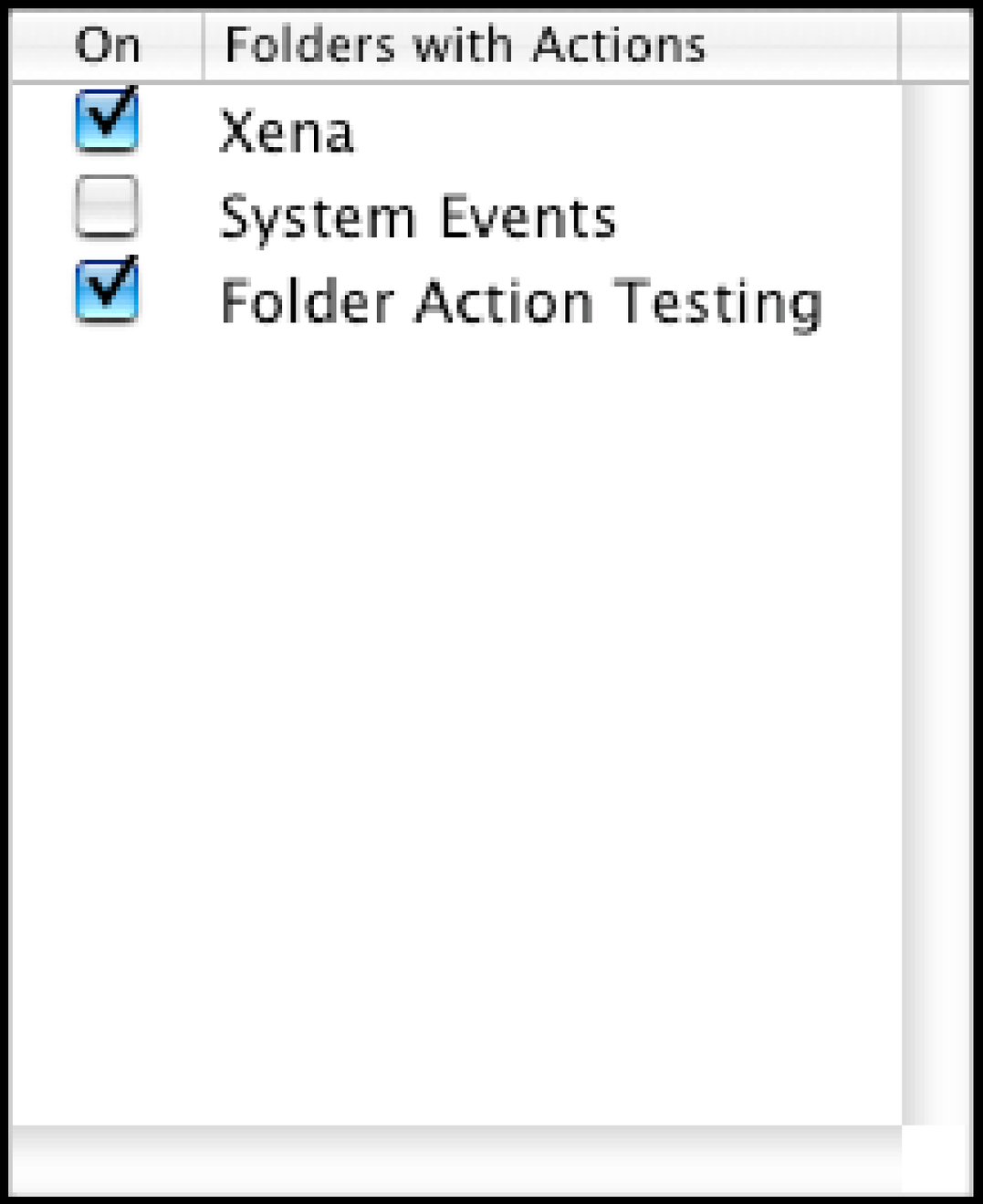


Table View

Conceptual mapping

- `application`
- `folder action`
  - By index, by name
  - Properties
    - `name`
    - `enabled`

Designing a Dictionary

Summary

- Use an object model
- Define verbs only as necessary
- Omit: a, an, the, me, my, is
- Don’t use InterCaps, use white space instead
- Use lower case, except for acronyms and proper nouns
- Reuse terminology from Registries and other applications, shamelessly
- Plan for a full object model
- Implement as much as you can in the first release

Implementing the Objects

Three Scenarios

Implementing the Objects

Three scenarios

- Starting from scratch
  - Best case
  - Maximum freedom
- Adding to existing objects
  - Typical case
  - Some simple retrofitting required
- Building alongside existing objects
  - Worst case
  - Extensive, complex retrofitting may be required

Implementing the Objects

Starting from scratch

- Where did my dictionary come from?
  - Marketing requirements
  - Competitive analysis
  - Design review
    - Marketeers, engineers, testers, tech writers, customers
  - Prototyping
- Consider the resulting AppleScript dictionary to be your design document for the entire application

Implementing the Objects

Implement a scriptable base class

- Leverage a reusable object
- Consistent behavior across objects
- Consistent behavior across application suites
- Less code is less bugs

Implementing the Objects

Implement a scriptable base class

- Mine is called `Element`
- `-name`, `-setName:`
- `-uniqueID`, `-setUniqueID:`
  - Serial number
  - Time stamp
  - UUID

Implementing the Objects

Implement a scriptable base class

- `–objectSpecifier`
  - If `–uniqueID` is not `nil`, and is in the dictionary, use it
    - Must be persistent throughout a launch
  - If `–name` is not `nil`, and is in the dictionary, use it
  - Otherwise, use an index specifier

Implementing the Objects

Implement a scriptable base class

- `–parent`, `–setParent:`
  - Do NOT retain
- `–siblings`, `–setSiblings:`
  - Returns `@"elementName"`
    - The method name for accessing this class of element
  - Cocoa Scripting derives the object's class
- `–attributes`, `–setAttributes:`
  - An NSDictionary of property values
  - Not shown in the AppleScript dictionary

Implementing the Objects

Implement your scriptable objects

- Inherit from your scriptable base class
- Add an `-init` method with no parameters
  - Used by `make new`
  - Chain to base class' `-init` method
  - Object must be ready to use at this point
- Add a `-dealloc` method
  - Release any owned objects
  - Chain to base class' `-dealloc` method

Implementing the Objects

Implement your scriptable objects

- Add –*propertyName* and –set*PropertyName*: for each property
  - Return plausible default values
- Add –*elementName* for each element relationship
  - Add just enough code to test containment
  - Fake it, if necessary
- Extend NSApplication
  - Use an Objective-C category or delegate to add methods
  - Use globals to add persistent properties and elements

Implementing the Objects

Implement your scriptable objects

- Add `-insertIn`*`ElementName`*`:atIndex:` and `-insertIn`*`ElementName`*`:` for each element relationship
  - Used by `make new`
- Add `-removeFrom`*`ElementName`*`AtIndex:` for each element relationship
  - Used by `delete`

Implementing the Objects

Implement your scriptable objects

- Enhanced element accessors can improve performance
  - See last year's session for details
    - WWDC 2002 Session 303 Cocoa Scripting
  - http://developer.apple.com/techpubs/macosx/Cocoa/Reference/Foundation/ObjC_classic/Protocols/NSScriptKeyValueCoding.html
  - Cocoa Scripting Release Notes
    ADC Home > Documentation > AppleScript >

Testing the Objects

Testing your code as you write it

Testing the Objects

Make your scriptable objects self-diagnostic

- Add a logging macro to each method

```
#define jgcLogSwitch 0
#if jgcLogSwitch
#define jgcLog(x) NSLog(x)
#else
#define jgcLog(x)
#end
…
jgcLog(@”JGC ObjectName methodName”);
```

Testing the Objects

Turn on system wide diagnostics

- Cocoa Scripting logging
  - `defaults write NSGlobalDomain NSScriptingDebugLogLevel 1`
- Apple Event Manager logging
  - `setenv AEDebug 1`
    `setenv AEDebugSends 1`
    `setenv AEDebugReceives 1`
    `setenv AEDebugVerbose 1`
  - Launch your application in Terminal

Testing the Objects

Write your first script

- Probe the entire hierarchy

```
tell application “System Events”
  get properties
  get every folder action
  get properties of every folder action
  get every script ¬
    of every folder action
  get properties of every script ¬
    of every folder action
end tell
```

Implementing the Objects

Completing the implementation

- Empower your scriptable objects
  - Update *–propertyName* and –setPropertyName: for each property
    - Return real data from APIs, files, Web, etc.
  - Update *–elementName* for each element relationship
    - Return an array of real objects with real data
- Rerun your test script(s)

Testing the Objects

Write lots more scripts

- Test access by index

```
tell application “System Events”
  set m to number of folder actions
  repeat with j from 1 to m
    tell folder action j
      get properties
      get every script -- loop here, too
    end tell
  end repeat
end tell
```

Testing the Objects

Write lots more scripts

- Test access by name

```
tell application “System Events”
  set x to name of every folder action
  repeat with y in x
    tell folder action y
      get properties
      get every script -- loop here, too
    end tell
  end repeat
end tell
```

Testing the Objects

Write lots more scripts

- Test access by unique ID

```
tell application “System Events”
  set x to id of every folder action
  repeat with y in x
    tell folder action id y
      get properties
      get every script -- loop here, too
    end tell
  end repeat
end tell
```

Testing the Objects

Write lots more scripts

- Test access by object specifier

```
tell application “System Events”
  set x to every folder action
  repeat with y in x
    tell y
      get properties
      get every script -- loop here, too
    end tell
  end repeat
end tell
```

Testing the Objects

Write lots more scripts

- Make something new

```
tell application “System Events”
  if exists folder action “Desktop” then
    delete folder action “Desktop”
  end if
  make new folder action ¬
    with properties {path:”~/Desktop”}
end tell
```

- Watch how this works with the logging turned on, it’s fascinating

Testing the Objects

Write lots more scripts

- Whose clauses are very important

```
tell application “System Events”
  get every folder action ¬
    whose enabled is true
  get every script ¬
    of every folder action ¬
    whose enabled is true
  get every script ¬
    of (every folder action ¬
    whose enabled is true)
end tell
```

Other Scenarios

Scripting existing applications

Implementing the Objects

Adding to existing objects

- Define scriptability for your objects
  - Use Objective-C categories
    - Powerful and non-intrusive
  - Add or reuse an `-init` method with no parameters
    - Object must be ready to use at this point
  - Add or reuse a `-dealloc` method

Implementing the Objects

Adding to existing objects

- Define scriptability for your objects
  - Add or reuse `—propertyName` and `—setPropertyName:` methods
  - Add or reuse `—elementName` and associated efficiency methods
  - Extend NSApplication
    - Use an Objective-C category or delegate to add methods
    - Use globals to add persistent properties and elements
  - Add a logging macro to each method

Testing the Objects

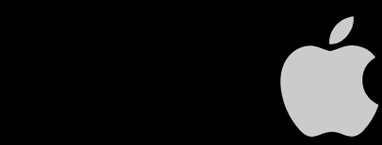
Adding to existing objects

- Write lots of scripts
- Empower your scriptable objects
  - Update `–propertyName` and `–setPropertyName:` methods
  - Update `–elementName` and associated efficiency methods
- Rerun your test scripts

Implementing and Testing the Objects

Building alongside existing objects

- Define your scriptable objects
  - Define new objects, properties, and element relationships, as if starting from scratch
- Write lots of scripts
- Empower your scriptable objects
  - Tie into existing APIs or objects to access data
  - This is where the complexity can arise
- Rerun your scripts



Hey, What About Carbon?

- Scripting a Carbon application is still the same
- http://developer.apple.com/techpubs/mac/IAC/IAC-2.html
- http://www.mactech.com/articles/develop/issue_28/reuter.html
- Seriously consider migrating to Cocoa
  - Objective-C and Cocoa Scripting make your work easier

Can I Use Cocoa Scripting with Carbon?

- It is possible to use Cocoa Scripting in an otherwise Carbon application
- Not too hard, but not trivial, either
- Can have a measurable performance impact
  - Launch time
  - Memory usage
- To try this, contact Developer Technical Support
  - dts@apple.com
  - Mention “scripting hybrid” in the subject

Documentation

Where to find more information

Documentation

Designing a scriptable application

- Topic: Application Architecture
  - Overview, concepts
  - Links to related class reference
- Documentation>Mac OS X>Cocoa>Program Design
- http://developer.apple.com/techpubs/macosx/Cocoa/CocoaTopics.html

Documentation

Implementing a scriptable application

- Topic: Scriptable Applications
  - Overview, concepts, programming tasks
  - Links to scripting class reference
  - Links to other AppleScript documentation

- Documentation>Mac OS X>Cocoa>Program Design
- http://developer.apple.com/techpubs/macosx/Cocoa/CocoaTopics.html

Reference Library

Documentation

AppleScript Language Guide

ADC Home > Documentation > AppleScript >

Scriptable Cocoa Applications

ADC Home > Documentation > AppleScript >

Carbon Specification for Open Scripting Architecture

ADC Home > Documentation > AppleScript >

Reference Library

Documentation (continued)

AppleScript Release Notes

ADC Home > Documentation > AppleScript >

Cocoa Scripting Release Notes

ADC Home > Documentation > AppleScript >

Technical Notes

Interapplication Communication Technical Notes

ADC Home > Technical Notes > Interapplication Communications

Reference Library

Technical Q&As

AppleScript Q&As

ADC Home > Technical Q&As > AppleScript

Sample Code

Interapplication Communications

ADC Home > Sample Code > Interapplication Comm

Reference Library

Other Resources

AppleScript Website

http://www.apple.com/applescript/

“AppleScript in a Nutshell”

Published by O’Reilly & Associates

Other third-party books on AppleScript

Available in technical bookstores and online

Third-party AppleScript websites

Many available

Roadmap

401 AppleScript Update	Russian Hill Mon., 5:00pm
FF005 AppleScript Feedback Forum	North Beach Wed., 9:00am
718 AppleScript and QuickTime	SOMA Wed., 2:00pm
306 AppleScript Studio	Pacific Heights Wed., 5:00pm

Roadmap

414 AppleScript: Making Your Carbon and Cocoa Application Scriptable	The Presidio Thu., 10:30am
311 Software Testing Tools for Mac OS X	Russian Hill Fri., 10:30am
623 AppleScript for System Administrators	Pacific Heights Fri., 2:00pm

Who to Contact

Todd Fernandez
Manager, AppleScript Engineering
trf@apple.com

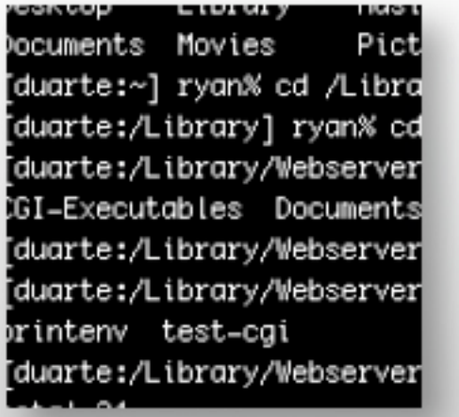
Sal Soghoian
AppleScript Product Manager
sal@apple.com

Jason Yeo
Mac OS X Technology Manager
jason@apple.com

John Montbriand
AppleScript DTS Engineer
montbriand@apple.com

http://developer.apple.com/wwdc2003/urls.html

Q&A



Jason Yeo

Mac OS X Technology Manager

jason@apple.com

http://developer.apple.com/wwdc2003/urls.html

WWDC 2003

Worldwide Developers Conference